

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

APPLICATION FOR LETTERS PATENT

**EV316935392**

**Administrative Tool Environment**

Inventor(s):  
Jeffrey P. Snover  
Daryl W. Wray  
James W. Truher III  
Bruce G. Payette

ATTORNEY'S DOCKET NO. MS1-1737US

## **TECHNICAL FIELD**

Subject matter disclosed herein relates to runtime environments, and in particular to administrative tool environments for automating administrative tasks.

## **BACKGROUND OF THE INVENTION**

Administrative tasks support the day to day operation of computing devices, such as laptop, desktops, and the like. For example, administrative tasks typically allow a user to create a new folder, install or upgrade an application, and perform various other system tasks. System administrators, who maintain networked computers, perform even more complex administrative tasks, such as creating new users, deploying software and software patches, monitoring the network, troubleshooting the network, and the like. Often, command line interfaces or graphical user interfaces (GUI) facilitate execution of administrative tasks.

Most graphical user interfaces require the user to navigate a series of menus or pages and then click on a desired option. According to such interfaces, the user is required to follow the series of menus or pages and is not allowed to jump from one page to another without starting over or traversing up and back down the series of pages. Such interfaces may be referred to as restricted navigation interfaces wherein the user must follow a regimented and predetermined procedure to achieve an administrative goal. A novice user may benefit from using such a GUI because it provides helpful information on each menu or page that directs the novice user to the desired administrative task. However, for many advanced users, such as system administrators, using a GUI that has rigid navigation

1 restrictions is very cumbersome. These advanced users already know the  
2 administrative task that they want to perform. In addition, advance users desire at  
3 least some degree of task automation. However, automating administrative tasks  
4 accessed through a GUI is very difficult. Also, if the GUI changes, the  
5 corresponding automation may no longer operate properly. Thus, many advanced  
6 users prefer using a more robust command line interface.

7 A command line interface allows a user to directly perform a task by typing  
8 in a command. One disadvantage with a command line interface is that the user  
9 must know the exact command to type in because helpful information is not  
10 provided on the display. However, once the command is known, it becomes much  
11 more efficient to type the command than to navigate a series of menus. In  
12 addition, tasks executed from a command line interface may be automated. For  
13 example, when users notice that they are continually typing the same commands  
14 (e.g., a series of commands, etc.) they may create a script (e.g., a .bat file) that  
15 includes the repetitive commands. When the script is executed, all the commands  
16 listed in the script are executed. The users may also soon notice that different  
17 scripts contain similar items. Upon noticing this, the users may wish to create  
18 parameterized subroutines within the script to further automate the tasks.

19 However, many scripting solutions that provide sophisticated automation  
20 features (e.g., parameterized subroutines) are developed in the same manner as a  
21 full blown application and require system programming knowledge. Thus, these  
22 solutions are not ideal for system administrators who do not generally have  
23 sophisticated programming knowledge.  
24  
25

1 Therefore, currently, system administrators must either forego automation  
2 or automate using systems programming skills. Neither option is highly desirable.  
3 If the administrator selects to forego automation, additional costs are incurred to  
4 support several other administrators to help run the tasks manually. However, if  
5 the administrator selects to automate the tasks, costs are incurred to support a  
6 programming environment and to obtain the necessary programming skills. In  
7 addition, maintenance costs are expensive because the automated tool must be  
8 rebuilt whenever a change occurs.

9 Therefore, there is a need for an administrative tool framework that allows  
10 administrators to effectively and efficiently automate administrative tasks.

## 11 **SUMMARY OF THE INVENTION**

12 An administrative tool framework is provided. User input is supplied to the  
13 administrative tool framework for processing. The administrative tool framework  
14 maps user input to cmdlet objects. The cmdlet object is associated with a  
15 grammar for parsing the user input and input objects to obtain expected input  
16 parameters identified within the cmdlet. The grammar may be associated directly  
17 within the cmdlet, such as using parameter declarations. Alternatively, the  
18 grammar may be associated indirectly with the cmdlet, such through an XML  
19 document. The input objects are emitted by one cmdlet and are available as input  
20 to another cmdlet. The input objects may be any precisely parseable input, such as  
21 .NET objects, plain strings, XML documents, and the like. The cmdlets may  
22 operate within the same process. Alternatively, one cmdlet may operate locally  
23 while another cmdlet operates in another process or remotely. The cmdlets may  
24 be provided by the administrative tool framework or may be provided by third  
25

1 party developers. The user input may be supplied to the framework via a host  
2 cmdlet.

### 3 4 **BRIEF DESCRIPTION OF THE DRAWINGS**

5 FIGURE 1 illustrates an exemplary computing device that may use an  
6 exemplary administrative tool environment.

7 FIGURE 2 is a block diagram generally illustrating an overview of an  
8 exemplary administrative tool framework for the present administrative tool  
9 environment.

10 FIGURE 3 is a block diagram illustrating components within the host-  
11 specific components of the administrative tool framework shown in FIGURE 2.

12 FIGURE 4 is a block diagram illustrating components within the core  
13 engine component of the administrative tool framework shown in FIGURE 2.

14 FIGURE 5 is one exemplary data structure for specifying a cmdlet suitable  
15 for use within the administrative tool framework shown in FIGURE 2.

16 FIGURE 6 is an exemplary data structure for specifying a command base  
17 type from which a cmdlet shown in FIGURE 5 is derived.

18 FIGURE 7 is another exemplary data structure for specifying a cmdlet  
19 suitable for use within the administrative tool framework shown in FIGURE 2.

20 FIGURE 8 is a logical flow diagram illustrating an exemplary process for  
21 host processing that is performed within the administrative tool framework shown  
22 in FIGURE 2.

23 FIGURE 9 is a logical flow diagram illustrating an exemplary process for  
24 handling input that is performed within the administrative tool framework shown  
25 in FIGURE 2.

1       FIGURE 10 is a logical flow diagram illustrating a process for processing  
2 scripts suitable for use within the process for handling input shown in FIGURE 9.

3       FIGURE 11 is a logical flow diagram illustrating a script pre-processing  
4 process suitable for use within the script processing process shown in FIGURE 10.

5       FIGURE 12 is a logical flow diagram illustrating a process for applying  
6 constraints suitable for use within the script processing process shown in FIGURE  
7 10.

8       FIGURE 13 is a functional flow diagram illustrating the processing of a  
9 command string in the administrative tool framework shown in FIGURE 2.

10       FIGURE 14 is a logical flow diagram illustrating a process for processing  
11 commands strings suitable for use within the process for handling input shown in  
12 FIGURE 9.

13       FIGURE 15 is a logical flow diagram illustrating an exemplary process for  
14 creating an instance of a cmdlet suitable for use within the processing of command  
15 strings shown in FIGURE 14.

16       FIGURE 16 is a logical flow diagram illustrating an exemplary process for  
17 populating properties of a cmdlet suitable for use within the processing of  
18 commands shown in FIGURE 14.

19       FIGURE 17 is a logical flow diagram illustrating an exemplary process for  
20 executing the cmdlet suitable for use within the processing of commands shown in  
21 FIGURE 14.

22       FIGURE 18 is a functional block diagram of an exemplary extended type  
23 manager suitable for use within the administrative tool framework shown in  
24 FIGURE 2.

1       FIGURE 19 graphically depicts exemplary sequences for output processing  
2 cmdlets within a pipeline.

3       FIGURE 20 illustrates exemplary processing performed by one of the  
4 output processing cmdlets shown in FIGURE 19.

5       FIGURE 21 graphically depicts an exemplary structure for display  
6 information accessed during the processing of FIGURE 20.

7       FIGURE 22 is a table listing an exemplary syntax for exemplary output  
8 processing cmdlets.

9       FIGURE 23 illustrates results rendered by the out/console cmdlet using  
10 various pipeline sequences of the output processing cmdlets.

## 11 12 **DETAILED DESCRIPTION**

13       Briefly stated, the present administrative tool environment provides a  
14 computing framework that reduces administrative costs associated with  
15 maintaining and supporting computers and computer networks. In addition, the  
16 framework supports rapid ad-hoc development of administrative tools by both  
17 programmers and non-programmers.

18       The following description sets forth a specific exemplary administrative  
19 tool environment. Other exemplary environments may include features of this  
20 specific embodiment and/or other features, which aim to facilitate administrative  
21 tasks.

22       The following detailed description is divided into several sections. A first  
23 section describes an illustrative computing environment in which the  
24 administrative tool environment may operate. A second section describes an  
25 exemplary framework for the administrative tool environment. Subsequent

1 sections describe individual components of the exemplary framework and the  
2 operation of these components.

### 3 Exemplary Computing Environment

4 FIGURE 1 illustrates an exemplary computing device that may be used in  
5 an exemplary administrative tool environment. In a very basic configuration,  
6 computing device 100 typically includes at least one processing unit 102 and  
7 system memory 104. Depending on the exact configuration and type of  
8 computing device, system memory 104 may be volatile (such as RAM), non-  
9 volatile (such as ROM, flash memory, etc.) or some combination of the two.  
10 System memory 104 typically includes an operating system 105, one or more  
11 program modules 106, and may include program data 107. The operating system  
12 106 include a component-based framework 120 that supports components  
13 (including properties and events), objects, inheritance, polymorphism, reflection,  
14 and provides an object-oriented component-based application programming  
15 interface (API), such as that of the .NET™ Framework manufactured by  
16 Microsoft Corporation, Redmond, WA. The operating system 105 also includes  
17 an administrative tool framework 200 that interacts with the component-based  
18 framework 120 to support development of administrative tools (not shown). This  
19 basic configuration is illustrated in FIGURE 1 by those components within dashed  
20 line 108.

21 Computing device 100 may have additional features or functionality. For  
22 example, computing device 100 may also include additional data storage devices  
23 (removable and/or non-removable) such as, for example, magnetic disks, optical  
24 disks, or tape. Such additional storage is illustrated in FIGURE 1 by removable  
25



1 storage 109 and non-removable storage 110. Computer storage media may  
2 include volatile and nonvolatile, removable and non-removable media  
3 implemented in any method or technology for storage of information, such as  
4 computer readable instructions, data structures, program modules, or other data.  
5 System memory 104, removable storage 109 and non-removable storage 110 are  
6 all examples of computer storage media. Computer storage media includes, but is  
7 not limited to, RAM, ROM, EEPROM, flash memory or other memory  
8 technology, CD-ROM, digital versatile disks (DVD) or other optical storage,  
9 magnetic cassettes, magnetic tape, magnetic disk storage or other magnetic storage  
10 devices, or any other medium which can be used to store the desired information  
11 and which can be accessed by computing device 100. Any such computer storage  
12 media may be part of device 100. Computing device 100 may also have input  
13 device(s) 112 such as keyboard, mouse, pen, voice input device, touch input  
14 device, etc. Output device(s) 114 such as a display, speakers, printer, etc. may  
15 also be included. These devices are well know in the art and need not be  
16 discussed at length here.

17 Computing device 100 may also contain communication connections 116  
18 that allow the device to communicate with other computing devices 118, such as  
19 over a network. Communication connections 116 are one example of  
20 communication media. Communication media may typically be embodied by  
21 computer readable instructions, data structures, program modules, or other data in  
22 a modulated data signal, such as a carrier wave or other transport mechanism, and  
23 includes any information delivery media. The term "modulated data signal"  
24 means a signal that has one or more of its characteristics set or changed in such a  
25

1 manner as to encode information in the signal. By way of example, and not  
2 limitation, communication media includes wired media such as a wired network or  
3 direct-wired connection, and wireless media such as acoustic, RF, infrared and  
4 other wireless media. The term computer readable media as used herein includes  
5 both storage media and communication media.

### 6 Exemplary Administrative Tool Framework

7 FIGURE 2 is a block diagram generally illustrating an overview of an  
8 exemplary administrative tool framework 200. Administrative tool framework  
9 200 includes one or more host components 202, host-specific components 204,  
10 host-independent components 206, and handler components 208. The host-  
11 independent components 206 may communicate with each of the other  
12 components (i.e., the host components 202, the host-specific components 204, and  
13 the handler components 208). Each of these components are briefly described  
14 below and described in further detail, as needed, in subsequent sections.

### 15 Host components

16 The host components 202 include one or more host programs (e.g., host  
17 programs 210-214) that expose automation features for an associated application  
18 to users or to other programs. Each host program 210-214 may expose these  
19 automation features in its own particular style, such as via a command line, a  
20 graphical user interface (GUI), a voice recognition interface, application  
21 programming interface (API), a scripting language, a web service, and the like.  
22 However, each of the host programs 210-214 expose the one or more automation  
23 features through a mechanism provided by the administrative tool framework.  
24  
25

1 In this example, the mechanism uses cmdlets to surface the administrative  
2 tool capabilities to a user of the associated host program 210-214. In addition, the  
3 mechanism uses a set of interfaces made available by the host to embed the  
4 administrative tool environment within the application associated with the  
5 corresponding host program 210-214. Throughout the following discussion, the  
6 term "cmdlet" is used to refer to commands that are used within the exemplary  
7 administrative tool environment described with reference to FIGURES 2-23.

8 Cmdlets correspond to commands in traditional administrative  
9 environments. However, cmdlets are quite different than these traditional  
10 commands. For example, cmdlets are typically smaller in size than their  
11 counterpart commands because the cmdlets can utilize common functions  
12 provided by the administrative tool framework, such as parsing, data validation,  
13 error reporting, and the like. Because such common functions can be implemented  
14 once and tested once, the use of cmdlets throughout the administrative tool  
15 framework allows the incremental development and test costs associated with  
16 application-specific functions to be quite low compared to traditional  
17 environments.

18 In addition, in contrast to traditional environments, cmdlets do not need to  
19 be stand-alone executable programs. Rather, cmdlets may run in the same  
20 processes within the administrative tool framework. This allows cmdlets to  
21 exchange "live" objects between each other. This ability to exchange "live"  
22 objects allows the cmdlets to directly invoke methods on these objects. The  
23 details for creating and using cmdlets are described in further detail below.  
24  
25

1 In overview, each host program **210-214** manages the interactions between  
2 the user and the other components within the administrative tool framework.  
3 These interactions may include prompts for parameters, reports of errors, and the  
4 like. Typically, each host program **210-213** may provide its own set of specific  
5 host cmdlets (e.g., host cmdlets **218**). For example, if the host program is an email  
6 program, the host program may provide host cmdlets that interact with mailboxes  
7 and messages. Even though FIGURE 2 illustrates host programs **210-214**, one  
8 skilled in the art will appreciate that host components **202** may include other host  
9 programs associated with existing or newly created applications. These other host  
10 programs will also embed the functionality provided by the administrative tool  
11 environment within their associated application. The processing provided by a  
12 host program is described in detail below in conjunction with FIGURE 8.

13 In the examples illustrated in FIGURE 2, a host program may be a  
14 management console (i.e., host program **210**) that provides a simple, consistent,  
15 administration user interface for users to create, save, and open administrative  
16 tools that manage the hardware, software, and network components of the  
17 computing device. To accomplish these functions, host program **210** provides a  
18 set of services for building management GUIs on top of the administrative tool  
19 framework. The GUI interactions may also be exposed as user-visible scripts that  
20 help teach the users the scripting capabilities provided by the administrative tool  
21 environment.

22 In another example, the host program may be a command line interactive  
23 shell (i.e., host program **212**). The command line interactive shell may allow shell  
24  
25

1 metadata 216 to be input on the command line to affect processing of the  
2 command line.

3 In still another example, the host program may be a web service (i.e., host  
4 program 214) that uses industry standard specifications for distributed computing  
5 and interoperability across platforms, programming languages, and applications.

6 In addition to these examples, third parties may add their own host  
7 components by creating “third party” or “provider” interfaces and provider  
8 cmdlets that are used with their host program or other host programs. The  
9 provider interface exposes an application or infrastructure so that the application  
10 or infrastructure can be manipulated by the administrative tool framework. The  
11 provider cmdlets provide automation for navigation, diagnostics, configuration,  
12 lifecycle, operations, and the like. The provider cmdlets exhibit polymorphic  
13 cmdlet behavior on a completely heterogeneous set of data stores. The  
14 administrative tool environment operates on the provider cmdlets with the same  
15 priority as other cmdlet classes. The provider cmdlet is created using the same  
16 mechanisms as the other cmdlets. The provider cmdlets expose specific  
17 functionality of an application or an infrastructure to the administrative tool  
18 framework. Thus, through the use of cmdlets, product developers need only create  
19 one host component that will then allow their product to operate with many  
20 administrative tools. For example, with the exemplary administrative tool  
21 environment, system level graphical user interface help menus may be integrated  
22 and ported to existing applications.

### 23 Host-specific components

24  
25

1       The host-specific components **204** include a collection of services that  
2       computing systems (e.g., computing device **100** in FIGURE 1) use to isolate the  
3       administrative tool framework from the specifics of the platform on which the  
4       framework is running. Thus, there is a set of host-specific components for each  
5       type of platform. The host-specific components allow the users to use the same  
6       administrative tools on different operating systems.

7       Turning briefly to FIGURE 3, the host-specific components **204** may  
8       include an intellisense/metadata access component **302**, a help cmdlet component  
9       **304**, a configuration/registration component **306**, a cmdlet setup component **308**,  
10      and an output interface component **309**. Components **302-308** communicate with a  
11      database store manager **312** associated with a database store **314**. The parser **220**  
12      and script engine **222** communicate with the intellisense/metadata access  
13      component **302**. The core engine **224** communicates with the help cmdlet  
14      component **304**, the configuration/registration component **306**, the cmdlet setup  
15      component **308**, and the output interface component **309**. The output interface  
16      component **309** includes interfaces provided by the host to out cmdlets. These out  
17      cmdlets can then call the host's output object to perform the rendering. Host-  
18      specific components **204** may also include a logging/auditing component **310**,  
19      which the core engine **224** uses to communicate with host specific (i.e., platform  
20      specific) services that provide logging and auditing capabilities.

21      In one exemplary administrative tool framework, the intellisense/metadata  
22      access component **302** provides auto-completion of commands, parameters, and  
23      parameter values. The help cmdlet component **304** provides a customized help  
24      system based on a host user interface.  
25

## **Handler components**

Referring back to FIGURE 2, the handler components **208** includes legacy utilities **230**, management cmdlets **232**, non-management cmdlets **234**, remoting cmdlets **236**, and a web service interface **238**. The management cmdlets **232** (also referred to as platform cmdlets) include cmdlets that query or manipulate the configuration information associated with the computing device. Because management cmdlets **232** manipulate system type information, they are dependant upon a particular platform. However, each platform typically has management cmdlets **232** that provide similar actions as management cmdlets **232** on other platforms. For example, each platform supports management cmdlets **232** that get and set system administrative attributes (e.g., get/process, set/IPAddress). The host-independent components **206** communicate with the management cmdlets via cmdlet objects generated within the host-independent components **206**. Exemplary data structures for cmdlets objects will be described in detail below in conjunction with FIGURES 5-7.

The non-management cmdlets **234** (sometimes referred to as base cmdlets) include cmdlets that group, sort, filter, and perform other processing on objects provided by the management cmdlets **232**. The non-management cmdlets **234** may also include cmdlets for formatting and outputting data associated with the pipelined objects. An exemplary mechanism for providing a data driven command line output is described below in conjunction with FIGURES 19-23. The non-management cmdlets **234** may be the same on each platform and provide a set of utilities that interact with host-independent components **206** via cmdlet objects. The interactions between the non-management cmdlets **234** and the host-

1 independent components **206** allow reflection on objects and allow processing on  
2 the reflected objects independent of their (object) type. Thus, these utilities allow  
3 developers to write non-management cmdlets once and then apply these non-  
4 management cmdlets across all classes of objects supported on a computing  
5 system. In the past, developers had to first comprehend the format of the data that  
6 was to be processed and then write the application to process only that data. As a  
7 consequence, traditional applications could only process data of a very limited  
8 scope. One exemplary mechanism for processing objects independent of their  
9 object type is described below in conjunction with FIGURE 18.

10 The legacy utilities **230** include existing executables, such as win32  
11 executables that run under cmd.exe. Each legacy utility **230** communicates with  
12 the administrative tool framework using text streams (i.e., stdin and stdout), which  
13 are a type of object within the object framework. Because the legacy utilities **230**  
14 utilize text streams, reflection-based operations provided by the administrative tool  
15 framework are not available. The legacy utilities **230** execute in a different  
16 process than the administrative tool framework. Although not shown, other  
17 cmdlets may also operate out of process.

18 The remoting cmdlets **236**, in combination with the web service interface  
19 **238**, provide remoting mechanisms to access interactive and programmatic  
20 administrative tool environments on other computing devices over a  
21 communication media, such as internet or intranet (e.g., internet/intranet **240**  
22 shown in FIGURE 2). In one exemplary administrative tool framework, the  
23 remoting mechanisms support federated services that depend on infrastructure that  
24 spans multiple independent control domains. The remoting mechanism allows  
25



1 scripts to execute on remote computing devices. The scripts may be run on a  
2 single or on multiple remote systems. The results of the scripts may be processed  
3 as each individual script completes or the results may be aggregated and processed  
4 en-masse after all the scripts on the various computing devices have completed.

5 For example, web service 214 shown as one of the host components 202  
6 may be a remote agent. The remote agent handles the submission of remote  
7 command requests to the parser and administrative tool framework on the target  
8 system. The remoting cmdlets serve as the remote client to provide access to the  
9 remote agent. The remote agent and the remoting cmdlets communicate via a  
10 parsed stream. This parsed stream may be protected at the protocol layer, or  
11 additional cmdlets may be used to encrypt and then decrypt the parsed stream.

## 12 **Host-independent components**

13 The host-independent components 206 include a parser 220, a script engine  
14 222 and a core engine 224. The host-independent components 206 provide  
15 mechanisms and services to group multiple cmdlets, coordinate the operation of  
16 the cmdlets, and coordinate the interaction of other resources, sessions, and jobs  
17 with the cmdlets.

## 18 **Exemplary Parser**

19 The parser 220 provides mechanisms for receiving input requests from  
20 various host programs and mapping the input requests to uniform cmdlet objects  
21 that are used throughout the administrative tool framework, such as within the  
22 core engine 224. In addition, the parser 220 may perform data processing based  
23 on the input received. One exemplary method for performing data processing  
24 based on the input is described below in conjunction with FIGURE 12. The  
25

1 parser 220 of the present administrative tool framework provides the capability to  
2 easily expose different languages or syntax to users for the same capabilities. For  
3 example, because the parser 220 is responsible for interpreting the input requests,  
4 a change to the code within the parser 220 that affects the expected input syntax  
5 will essentially affect each user of the administrative tool framework. Therefore,  
6 system administrators may provide different parsers on different computing  
7 devices that support different syntax. However, each user operating with the same  
8 parser will experience a consistent syntax for each cmdlet. In contrast, in  
9 traditional environments, each command implemented its own syntax. Thus, with  
10 thousands of commands, each environment supported several different syntax,  
11 usually many of which were inconsistent with each other.

## 12 Exemplary Script Engine

13 The script engine 222 provides mechanisms and services to tie multiple  
14 cmdlets together using a script. A script is an aggregation of command lines that  
15 share session state under strict rules of inheritance. The multiple command lines  
16 within the script may be executed either synchronously or asynchronously, based  
17 on the syntax provided in the input request. The script engine 222 has the ability  
18 to process control structures, such as loops and conditional clauses and to process  
19 variables within the script. The script engine also manages session state and gives  
20 cmdlets access to session data based on a policy (not shown).

## 21 Exemplary Core Engine

22 The core engine 224 is responsible for processing cmdlets identified by the  
23 parser 220. Turning briefly to FIGURE 4, an exemplary core engine 224 within  
24 the administrative tool framework 200 is illustrated. The exemplary core engine  
25

1 224 includes a pipeline processor 402, a loader 404, a metadata processor 406, an  
2 error & event handler 408, a session manager 410, and an extended type manager  
3 412.

#### 4 Exemplary Metadata Processor

5 The metadata processor 406 is configured to access and store metadata  
6 within a metadata store, such as database store 314 shown in FIGURE 3. The  
7 metadata may be supplied via the command line, within a cmdlet class definition,  
8 and the like. Different components within the administrative tool framework 200  
9 may request the metadata when performing their processing. For example, parser  
10 202 may request metadata to validate parameters supplied on the command line.

#### 11 Exemplary Error & Event Processor

12 The error & event processor 408 provides an error object to store  
13 information about each occurrence of an error during processing of a command  
14 line. For additional information about one particular error and event processor  
15 which is particularly suited for the present administrative tool framework, refer to  
16 U.S. Patent Application No. \_\_\_\_ / U.S. Patent No. \_\_\_\_, entitled "System and  
17 Method for Persisting Error Information in a Command Line Environment", which  
18 is owned by the same assignee as the present invention, and is incorporated here  
19 by reference.

#### 20 Exemplary Session Manager

21 The session manager 410 supplies session and state information to other  
22 components within the administrative tool framework 200. The state information  
23 managed by the session manager may be accessed by any cmdlet, host, or core  
24 engine via programming interfaces. These programming interfaces allow for the  
25 creation, modification, and deletion of state information.

## Exemplary Pipeline Processor and Loader

The loader **404** is configured to load each cmdlet in memory in order for the pipeline processor **402** to execute the cmdlet. The pipeline processor **402** includes a cmdlet processor **420** and a cmdlet manager **422**. The cmdlet processor **420** dispatches individual cmdlets. If the cmdlet requires execution on a remote, or a set of remote machines, the cmdlet processor **420** coordinates the execution with the remoting cmdlet **236** shown in FIGURE 2. The cmdlet manager **422** handles the execution of aggregations of cmdlets. The cmdlet manager **422**, the cmdlet processor **420**, and the script engine **222** (FIGURE 2) communicate with each other in order to perform the processing on the input received from the host program **210-214**. The communication may be recursive in nature. For example, if the host program provides a script, the script may invoke the cmdlet manager **422** to execute a cmdlet, which itself may be a script. The script may then be executed by the script engine **222**. One exemplary process flow for the core engine is described in detail below in conjunction with FIGURE 14.

## Exemplary Extended Type Manager

As mentioned above, the administrative tool framework provides a set of utilities that allows reflection on objects and allows processing on the reflected objects independent of their (object) type. The administrative tool framework **200** interacts with the component framework on the computing system (component framework **120** in FIGURE 1) to perform this reflection. As one skilled in the art will appreciate, reflection provides the ability to query an object and to obtain a type for the object, and then reflect on various objects and properties associated with that type of object to obtain other objects and/or a desired value.

1 Even though reflection provides the administrative tool framework 200 a  
2 considerable amount of information on objects, the inventors appreciated that  
3 reflection focuses on the type of object. For example, when a database datatable is  
4 reflected upon, the information that is returned is that the datatable has two  
5 properties: a column property and a row property. These two properties do not  
6 provide sufficient detail regarding the "objects" within the datatable. Similar  
7 problems arise when reflection is used on extensible markup language (XML) and  
8 other objects.

9 Thus, the inventors conceived of an extended type manager 412 that  
10 focuses on the usage of the type. For this extended type manager, the type of  
11 object is not important. Instead, the extended type manager is interested in  
12 whether the object can be used to obtain required information. Continuing with  
13 the above datatable example, the inventors appreciated that knowing that the  
14 datatable has a column property and a row property is not particularly interesting,  
15 but appreciated that one column contained information of interest. Focusing on  
16 the usage, one could associate each row with an "object" and associate each  
17 column with a "property" of that "object". Thus, the extended type manager 412  
18 provides a mechanism to create "objects" from any type of precisely parse-able  
19 input. In so doing, the extended type manager 412 supplements the reflection  
20 capabilities provided by the component-based framework 120 and extends  
21 "reflection" to any type of precisely parse-able input.

22 In overview, the extended type manager is configured to access precisely  
23 parse-able input (not shown) and to correlate the precisely parse-able input with a  
24 requested data type. The extended type manager 412 then provides the requested  
25 information to the requesting component, such as the pipeline processor 402 or

1 parser 220. In the following discussion, precisely parse-able input is defined as  
2 input in which properties and values may be discerned. Some exemplary precisely  
3 parse-able input include Windows Management Instrumentation (WMI) input,  
4 ActiveX Data Objects (ADO) input, eXtensible Markup Language (XML) input,  
5 and object input, such as .NET objects. Other precisely parse-able input may  
6 include third party data formats.

7       Turning briefly to FIGURE 18, a functional block diagram of an exemplary  
8 extended type manager for use within the administrative tool framework is shown.  
9 For explanation purposes, the functionality (denoted by the number “3” within a  
10 circle) provided by the extended type manager is contrasted with the functionality  
11 provided by a traditional tightly bound system (denoted by the number “1” within  
12 a circle) and the functionality provided by a reflection system (denoted by the  
13 number “2” within a circle). In the traditional tightly bound system, a caller **1802**  
14 within an application directly accesses the information (e.g., properties P1 and P2,  
15 methods M1 and M2) within object A. As mentioned above, the caller **1802** must  
16 know, a priori, the properties (e.g., properties P1 and P2) and methods (e.g.,  
17 methods M1 and M2) provided by object A at compile time. In the reflection  
18 system, generic code **1820** (not dependent on any data type) queries a system **1808**  
19 that performs reflection **1810** on the requested object and returns the information  
20 (e.g., properties P1 and P2, methods M1 and M2) about the object (e.g., object A)  
21 to the generic code **1820**. Although not shown in object A, the returned  
22 information may include additional information, such as vendor, file, date, and the  
23 like. Thus, through reflection, the generic code **1820** obtains at least the same  
24 information that the tightly bound system provides. The reflection system also  
25

1 allows the caller **1802** to query the system and get additional information without  
2 any a priori knowledge of the parameters.

3 In both the tightly bound systems and the reflection systems, new data  
4 types can not be easily incorporated within the operating environment. For  
5 example, in a tightly bound system, once the operating environment is delivered,  
6 the operating environment can not incorporate new data types because it would  
7 have to be rebuilt in order to support them. Likewise, in reflection systems, the  
8 metadata for each object class is fixed. Thus, incorporating new data types is not  
9 usually done.

10 However, with the present extended type manager new data types can be  
11 incorporated into the operating system. With the extended type manager **1822**,  
12 generic code **1820** may reflect on a requested object to obtain extended data types  
13 (e.g., object A') provided by various external sources, such as a third party objects  
14 (e.g., object A' and B), a semantic web **1832**, an ontology service **1834**, and the  
15 like. As shown, the third party object may extend an existing object (e.g., object  
16 A') or may create an entirely new object (e.g., object B).

17 Each of these external sources may register their unique structure within a  
18 type metadata **1840** and may provide code **1842**. When an object is queried, the  
19 extended type manager reviews the type metadata **1840** to determine whether the  
20 object has been registered. If the object is not registered within the type metadata  
21 **1840**, reflection is performed. Otherwise, extended reflection is performed. The  
22 code **1842** returns the additional properties and methods associated with the type  
23 being reflected upon. For example, if the input type is XML, the code **1842** may  
24 include a description file that describes the manner in which the XML is used to  
25 create the objects from the XML document. Thus, the type metadata **1840**

describes how the extended type manager 412 should query various types of precisely parse-able input (e.g., third party objects A' and B, semantic web 1832) to obtain the desired properties for creating an object for that specific input type and the code 1842 provides the instructions to obtain these desired properties. As a result, the extended type manager 412 provides a layer of indirection that allows "reflection" on all types of objects.

In addition to providing extended types, the extend type manager 412 provides additional query mechanisms, such as a property path mechanism, a key mechanism, a compare mechanism, a conversion mechanism, a globber mechanism, a property set mechanism, a relationship mechanism, and the like. Each of these query mechanisms, described below in the section "Exemplary Extended Type Manager Processing", provides flexibility to system administrators when entering command strings. Various techniques may be used to implement the semantics for the extended type manager. Three techniques are described below. However, those skilled in the art will appreciate that variations of these techniques may be used without departing from the scope of the claimed invention.

In one technique, a series of classes having static methods (e.g., `getproperty()` ) may be provided. An object is input into the static method (e.g., `getproperty(object)` ), and the static method returns a set of results. In another technique, the operating environment envelopes the object with an adapter. Thus, no input is supplied. Each instance of the adapter has a `getproperty` method that acts upon the enveloped object and returns the properties for the enveloped object. The following is pseudo code illustrating this technique:



```

1      Class Adaptor
2      {
3          Object X;
4          getProperties();
5      }.

```

In still another technique, an adaptor class subclasses the object. Traditionally, subclassing occurred before compilation. However, with certain operating environments, subclassing may occur dynamically. For these types of environments, the following is pseudo code illustrating this technique:

```

12     Class Adaptor : A
13     {
14         getProperties()
15         {
16             return data;
17         }
18     }.

```

Thus, as illustrated in FIGURE 18, the extended type manager allows developers to create a new data type, register the data type, and allow other applications and cmdlets to use the new data type. In contrast, in prior administrative environments, each data type had to be known at compile time so that a property or method associated with an object instantiated from that data type could be directly accessed. Therefore, adding new data types that were supported by the administrative environment was seldom done in the past.

1 Referring back to FIGURE 2, in overview, the administrative tool  
2 framework 200 does not rely on the shell for coordinating the execution of  
3 commands input by users, but rather, splits the functionality into processing  
4 portions (e.g., host-independent components 206) and user interaction portions  
5 (e.g., via host cmdlets). In addition, the present administrative tool environment  
6 greatly simplifies the programming of administrative tools because the code  
7 required for parsing and data validation is no longer included within each  
8 command, but is rather provided by components (e.g., parser 220) within the  
9 administrative tool framework. The exemplary processing performed within the  
10 administrative tool framework is described below.

### 11 Exemplary Operation

12 FIGURES 5-7 graphically illustrate exemplary data structures used within  
13 the administrative tool environment. FIGURES 8-17 graphically illustrate  
14 exemplary processing flows within the administrative tool environment. One  
15 skilled in the art will appreciate that certain processing may be performed by a  
16 different component than the component described below without departing from  
17 the scope of the present invention. Before describing the processing performed  
18 within the components of the administrative tool framework, exemplary data  
19 structures used within the administrative tool framework are described.

### 20 Exemplary Data Structures for Cmdlet Objects

21 FIGURE 5 is an exemplary data structure for specifying a cmdlet suitable  
22 for use within the administrative tool framework shown in FIGURE 2. When  
23 completed, the cmdlet may be a management cmdlet, a non-management cmdlet, a  
24 host cmdlet, a provider cmdlet, or the like. The following discussion describes the  
25 creation of a cmdlet with respect to a system administrator's perspective (i.e., a

1 provider cmdlet). However, each type of cmdlet is created in the same manner  
2 and operates in a similar manner. A cmdlet may be written in any language, such  
3 as C#. In addition, the cmdlet may be written using a scripting language or the  
4 like. When the administrative tool environment operates with the .NET  
5 Framework, the cmdlet may be a .NET object.

6 The provider cmdlet **500** (hereinafter, referred to as cmdlet **500**) is a public  
7 class having a cmdlet class name (e.g., StopProcess **504**). Cmdlet **500** derives  
8 from a cmdlet class **506**. An exemplary data structure for a cmdlet class **506** is  
9 described below in conjunction with FIGURE 6. Each cmdlet **500** is associated  
10 with a command attribute **502** that associates a name (e.g., Stop/Process) with the  
11 cmdlet **500**. The name is registered within the administrative tool environment.  
12 As will be described below, the parser looks in the cmdlet registry to identify the  
13 cmdlet **500** when a command string having the name (e.g., Stop/Process) is  
14 supplied as input on a command line or in a script.

15 The cmdlet **500** is associated with a grammar mechanism that defines a  
16 grammar for expected input parameters to the cmdlet. The grammar mechanism  
17 may be directly or indirectly associated with the cmdlet. For example, the cmdlet  
18 **500** illustrates a direct grammar association. In this cmdlet **500**, one or more  
19 public parameters (e.g., ProcessName **510** and PID **512**) are declared. The  
20 declaration of the public parameters drives the parsing of the input objects to the  
21 cmdlet **500**. Alternatively, the description of the parameters may appear in an  
22 external source, such as an XML document. The description of the parameters in  
23 this external source would then drive the parsing of the input objects to the cmdlet.  
24  
25

Each public parameter **510**, **512** may have one or more attributes (i.e., directives ) associated with it. The directives may be from any of the following categories: parsing directive **521**, data validation directive **522**, data generation directive **523**, processing directive **524**, encoding directive **525**, and documentation directive **526**. The directives may be surrounded by square brackets. Each directive describes an operation to be performed on the following expected input parameter. Some of the directives may also be applied at a class level, such as user-interaction type directives. The directives are stored in the metadata associated with the cmdlet. The application of these attributes is described below in conjunction with FIGURE 12.

These attributes may also affect the population of the parameters declared within the cmdlet. One exemplary process for populating these parameters is described below in conjunction with FIGURE 16. The core engine may apply these directives to ensure compliance. The cmdlet **500** includes a first method **530** (hereinafter, interchangeably referred to as StartProcessing method **530**) and a second method **540** (hereinafter, interchangeably referred to as processRecord method **540**). The core engine uses the first and second methods **530**, **540** to direct the processing of the cmdlet **500**. For example, the first method **530** is executed once and performs set-up functions. The code **542** within the second method **540** is executed for each object (e.g., record) that needs to be processed by the cmdlet **500**. The cmdlet **500** may also include a third method (not shown) that cleans up after the cmdlet **500**.

Thus, as shown in FIGURE 5, code **542** within the second method **540** is typically quite brief and does not contain functionality required in traditional

1 administrative tool environments, such as parsing code, data validation code, and  
2 the like. Thus, system administrators can develop complex administrative tasks  
3 without learning a complex programming language.

4       FIGURE 6 is an exemplary data structure **600** for specifying a cmdlet base  
5 class **602** from which the cmdlet shown in FIGURE 5 is derived. The cmdlet base  
6 class **602** includes instructions that provide additional functionality whenever the  
7 cmdlet includes a hook statement and a corresponding switch is input on the  
8 command line or in the script (jointly referred to as command input).

9       The exemplary data structure **600** includes parameters, such as Boolean  
10 parameter verbose **610**, whatif **620**, and confirm **630**. As will be explained below,  
11 these parameters correspond to strings that may be entered on the command input.  
12 The exemplary data structure **600** may also include a security method **640** that  
13 determines whether the task being requested for execution is allowed.

14       FIGURE 7 is another exemplary data structure **700** for specifying a cmdlet.  
15 In overview, the data structure **700** provides a means for clearly expressing a  
16 contract between the administrative tool framework and the cmdlet. Similar to  
17 data structure 500, data structure **700** is a public class that derives from a cmdlet  
18 class **704**. The software developer specifies a cmdletDeclaration **702** that  
19 associates a noun/verb pair, such as "get/process" and "format/table", with the  
20 cmdlet **700**. The noun/verb pair is registered within the administrative tool  
21 environment. The verb or the noun may be implicit in the cmdlet name. Also,  
22 similar to data structure 500, data structure **700** may include one or more public  
23 members (e.g., Name **730**, Recurse **732**), which may be associated with the one or  
24 more directives 520-526 described in conjunction with data structure 500.

1        However, in this exemplary data structure 700, each of the expected input  
2 parameters 730 and 732 is associated with an input attribute 731 and 733,  
3 respectively. The input attributes 731 and 733 specifying that the data for its  
4 respective parameter 730 and 732 should be obtained from the command line.  
5 Thus, in this exemplary data structure 700, there are not any expected input  
6 parameters that are populated from a pipelined object that has been emitted by  
7 another cmdlet. Thus, data structure 700 does not override the first method (e.g.,  
8 StartProcessing) or the second method (e.g., ProcessRecord) which are provided  
9 by the cmdlet base class.

10        The data structure 700 may also include a private member 740 that is not  
11 recognized as an input parameter. The private member 740 may be used for  
12 storing data that is generated based on one of the directives.

13        Thus, as illustrated in data structure 700, through the use of declaring  
14 public properties and directives within a specific cmdlet class, cmdlet developers  
15 can easily specify a grammar for the expected input parameters to their cmdlets  
16 and specify processing that should be performed on the expected input parameters  
17 without requiring the cmdlet developers to generate any of the underlying logic.  
18 Data structure 700 illustrates a direct association between the cmdlet and the  
19 grammar mechanism. As mentioned above, this associated may also be indirect,  
20 such as by specifying the expected parameter definitions within an external source,  
21 such as an XML document.

22        The exemplary process flows within the administrative tool environment  
23 are now described.

## 24        **Exemplary Host Processing Flow**

FIGURE 8 is a logical flow diagram illustrating an exemplary process for host processing that is performed within the administrative tool framework shown in FIGURE 2. The process **800** begins at block **801**, where a request has been received to initiate the administrative tool environment for a specific application. The request may have been sent locally through keyboard input, such as selecting an application icon, or remotely through the web services interface of a different computing device. For either scenario, processing continues to block **802**.

At block **802**, the specific application (e.g., host program) on the “target” computing device sets up its environment. This includes determining which subsets of cmdlets (e.g., management cmdlets **232**, non-management cmdlets **234**, and host cmdlets **218**) are made available to the user. Typically, the host program will make all the non-management cmdlets **234** available and its own host cmdlets **218** available. In addition, the host program will make a subset of the management cmdlets **234** available, such as cmdlets dealing with processes, disk, and the like. Thus, once the host program makes the subsets of cmdlets available, the administrative tool framework is effectively embedded within the corresponding application. Processing continues to block **804**.

At block **804**, input is obtained through the specific application. As mentioned above, input may take several forms, such as command lines, scripts, voice, GUI, and the like. For example, when input is obtained via a command line, the input is retrieve from the keystrokes entered on a keyboard. For a GUI host, a string is composed based on the GUI. Processing continues at block **806**.

At block **806**, the input is provided to other components within the administrative tool framework for processing. The host program may forward the input directly to the other components, such as the parser. Alternatively, the host

1 program may forward the input via one of its host cmdlets. The host cmdlet may  
2 convert its specific type of input (e.g., voice) into a type of input (e.g., text string,  
3 script) that is recognized by the administrative tool framework. For example,  
4 voice input may be converted to a script or command line string depending on the  
5 content of the voice input. Because each host program is responsible for  
6 converting their type of input to an input recognized by the administrative tool  
7 framework, the administrative tool framework can accept input from any number  
8 of various host components. In addition, the administrative tool framework  
9 provides a rich set of utilities that perform conversions between data types when  
10 the input is forwarded via one of its cmdlets. Processing performed on the input  
11 by the other components is described below in conjunction with several other  
12 figures. Host processing continues at decision block **808**.

13 At decision block **808**, a determination is made whether a request was  
14 received for additional input. This may occur if one of the other components  
15 responsible for processing the input needs additional information from the user in  
16 order to complete its processing. For example, a password may be required to  
17 access certain data, confirmation of specific actions may be needed, and the like.  
18 For certain types of host programs (e.g., voice mail), a request such as this may  
19 not be appropriate. Thus, instead of querying the user for additional information,  
20 the host program may serialize the state, suspend the state, and send a notification  
21 so that at a later time the state may be resumed and the execution of the input be  
22 continued. In another variation, the host program may provide a default value  
23 after a predetermined time period. If a request for additional input is received,  
24 processing loops back to block **804**, where the additional input is obtained.  
25 Processing then continues through blocks **806** and **808** as described above. If no



1 request for additional input is received and the input has been processed,  
2 processing continues to block **810**.

3 At block **810**, results are received from other components within the  
4 administrative tool framework. The results may include error messages, status,  
5 and the like. The results are in an object form, which is recognized and processed  
6 by the host cmdlet within the administrative tool framework. As will be described  
7 below, the code written for each host cmdlet is very minimal. Thus, a rich set of  
8 output may be displayed without requiring a huge investment in development  
9 costs. Processing continues at block **812**.

10 At block **812**, the results may be viewed. The host cmdlet converts the  
11 results to the display style supported by the host program. For example, a returned  
12 object may be displayed by a GUI host program using a graphical depiction, such  
13 as an icon, barking dog, and the like. The host cmdlet provides a default format  
14 and output for the data. The default format and output may utilize the exemplary  
15 output processing cmdlets described below in conjunction with FIGURES 19-23.  
16 After the results are optionally displayed, the host processing is complete.

### 17 **Exemplary Process Flows for Handling Input**

18 FIGURE 9 is a logical flow diagram illustrating an exemplary process for  
19 handling input that is performed within the administrative tool framework shown  
20 in FIGURE 2. Processing begins at block **901** where input has been entered via a  
21 host program and forwarded to other components within the administrative tool  
22 framework. Processing continues at block **902**.

23 At block **902**, the input is received from the host program. In one  
24 exemplary administrative tool framework, the input is received by the parser,  
25

1 which deciphers the input and directs the input for further processing. Processing  
2 continues at decision block **904**.

3 At decision block **904**, a determination is made whether the input is a  
4 script. The input may take the form of a script or a string representing a command  
5 line (hereinafter, referred to as a “command string”). The command string may  
6 represent one or more cmdlets pipelined together. Even though the administrative  
7 tool framework supports several different hosts, each host provides the input as  
8 either a script or a command string for processing. As will be shown below, the  
9 interaction between scripts and command strings is recursive in nature. For  
10 example, a script may have a line that invokes a cmdlet. The cmdlet itself may be  
11 a script.

12 Thus, at decision block **904**, if the input is in a form of a script, processing  
13 continues at block **906**, where processing of the script is performed. Otherwise,  
14 processing continues at block **908**, where processing of the command string is  
15 performed. Once the processing performed within either block **906** or **908** is  
16 completed, processing of the input is complete.

### 17 **Exemplary Processing of Scripts**

18 FIGURE 10 is a logical flow diagram illustrating a process for processing a  
19 script suitable for use within the process for handling input shown in FIGURE 9.  
20 The process begins at block **1001**, where the input has been identified as a script.  
21 The script engine and parser communicate with each other to perform the  
22 following functions. Processing continues at block **1002**.

23 At block **1002**, pre-processing is performed on the script. Briefly, turning  
24 to FIGURE 11, a logical flow diagram is shown that illustrates a script pre-  
25

1 processing process **1100** suitable for use within the script processing process **1000**.

2 Script pre-processing begins at block **1101** and continues to decision block **1102**.

3 At decision block **1102**, a determination is made whether the script is being  
4 run for the first time. This determination may be based on information obtained  
5 from a registry or other storage mechanism. The script is identified from within  
6 the storage mechanism and the associated data is reviewed. If the script has not  
7 run previously, processing continues at block **1104**.

8 At block **1104**, the script is registered in the registry. This allows  
9 information about the script to be stored for later access by components within the  
10 administrative tool framework. Processing continues at block **1106**.

11 At block **1106**, help and documentation are extracted from the script and  
12 stored in the registry. Again, this information may be later accessed by  
13 components within the administrative tool framework. The script is now ready for  
14 processing and returns to block **1004** in FIGURE 10.

15 Returning to decision block **1102**, if the process concludes that the script  
16 has run previously, processing continues to decision block **1108**. At decision block  
17 **1108**, a determination is made whether the script failed during processing. This  
18 information may be obtained from the registry. If the script has not failed, the  
19 script is ready for processing and returns to block **1004** in FIGURE 10.

20 However, if the script has failed, processing continues at block **1110**. At  
21 block **1110**, the script engine may notify the user through the host program that the  
22 script has previously failed. This notification will allow a user to decide whether  
23 to proceed with the script or to exit the script. As mentioned above in conjunction  
24 with FIGURE 8, the host program may handle this request in various ways  
25 depending on the style of input (e.g., voice, command line). Once additional input

1 is received from the user, the script either returns to block **1004** in FIGURE 10 for  
2 processing or the script is aborted.

3       Returning to block **1004** in FIGURE 10, a line from the script is retrieved.  
4 Processing continues at decision block **1006**. At decision block **1006**, a  
5 determination is made whether the line includes any constraints. A constraint is  
6 detected by a predefined begin character (e.g., a bracket “[“) and a corresponding  
7 end character (e.g., a close bracket “]”). If the line includes constraints, processing  
8 continues to block **1008**.

9       At block **1008**, the constraints included in the line are applied. In general,  
10 the constraints provide a mechanism within the administrative tool framework to  
11 specify a type for a parameter entered in the script and to specify validation logic  
12 which should be performed on the parameter. The constraints are not only  
13 applicable to parameters, but are also applicable to any type of construct entered in  
14 the script, such as variables. Thus, the constraints provide a mechanism within an  
15 interpretive environment to specify a data type and to validate parameters. In  
16 traditional environments, system administrators are unable to formally test  
17 parameters entered within a script. An exemplary process for applying constraints  
18 is illustrated in FIGURE 12.

19       At decision block **1010**, a determination is made whether the line from the  
20 script includes built-in capabilities. Built-in capabilities are capabilities that are  
21 not performed by the core engine. Built-in capabilities may be processed using  
22 cmdlets or may be processed using other mechanisms, such as in-line functions. If  
23 the line does not have built-in capabilities, processing continues at decision block  
24 **1014**. Otherwise, processing continues at block **1012**.

1       At block **1012**, the built-in capabilities provided on the line of the script are  
2 processed. Example built-in capabilities may include execution of control  
3 structures, such as “if” statements, “for” loops, switches, and the like. Built-in  
4 capabilities may also include assignment type statements (e.g., a=3). Once the  
5 built-in capabilities have been processed, processing continues to decision block  
6 **1014**.

7       At decision block **1014**, a determination is made whether the line of the  
8 script includes a command string. The determination is based on whether the data  
9 on the line is associated with a command string that has been registered and with a  
10 syntax of the potential cmdlet invocation. As mentioned above, the processing of  
11 command strings and scripts may be recursive in nature because scripts may  
12 include command strings and command strings may execute a cmdlet that is a  
13 script itself. If the line does not include a command string, processing continues at  
14 decision block **1018**. Otherwise, processing continues at block **1016**.

15       At block **1016**, the command string is processed. In overview, the  
16 processing of the command string includes identifying a cmdlet class by the parser  
17 and passing the corresponding cmdlet object to the core engine for execution. The  
18 command string may also include a pipelined command string that is parsed into  
19 several individual cmdlet objects and individually processed by the core engine.  
20 One exemplary process for processing command strings is described below in  
21 conjunction with FIGURE 14. Once the command string is processed, processing  
22 continues at decision block **1018**.

23       At decision block **1018**, a determination is made whether there is another  
24 line in the script. If there is another line in the script, processing loops back to  
25

1 block **1004** and proceeds as described above in blocks **1004-1016**. Otherwise,  
2 processing is complete.

3 An exemplary process for applying constraints in block **1008** is illustrated  
4 in FIGURE 12. The process begins at block **1201** where a constraint is detected in  
5 the script or in the command string on the command line. When the constraint is  
6 within a script, the constraints and the associated construct may occur on the same  
7 line or on separate lines. When the constraint is within a command string, the  
8 constraint and the associated construct occur before the end of line indicator (e.g.,  
9 enter key). Processing continues to block **1202**.

10 At block **1202**, constraints are obtained from the interpretive environment.  
11 In one exemplary administrative tool environment, the parser deciphers the input  
12 and determines the occurrence of constraints. Constraints may be from one of the  
13 following categories: predicate directive, parsing directive, data validation  
14 directive, data generation directive, processing directive, encoding directive, and  
15 documentation directive. For one exemplary parsing syntax, the directives are  
16 surrounded by square brackets and describe the construct that follows them. The  
17 construct may be a function, a variable, a script, or the like.

18 As will be described below, through the use of directives, script authors are  
19 allowed to easily type and perform processing on the parameters within the script  
20 or command line (i.e., an interpretive environment) without requiring the script  
21 authors to generate any of the underlying logic. Processing continues to block  
22 **1204**.

23 At block **1204**, the constraints that are obtained are stored in the metadata  
24 for the associated construct. The associated construct is identified as being the  
25

1 first non-attribution token after one or more attribution tokens (tokens that denote  
2 constraints) have been encountered. Processing continues to block 1206.

3 At block 1206, whenever the construct is encountered within the script or in  
4 the command string, the constraints defined within the metadata are applied to the  
5 construct. The constraints may include data type, predicate directives 1210,  
6 documentation directives 1212, parsing directives 1214, data generation directives  
7 1216, data validation directives 1218, and object processing and encoding  
8 directives 1220. Constraints specifying data types may specify any data type  
9 supported by the system on which the administrative tool framework is running.  
10 Predicate directives 1210 are directives that indicate whether processing should  
11 occur. Thus, predicate directives 1210 ensure that the environment is correct for  
12 execution. For example, a script may include the following predicate directive:

13 [PredicateScript("isInstalled","ApplicationZ")].  
14

15 The predicate directive ensures that the correct application is installed on  
16 the computing device before running the script. Typically, system environment  
17 variables may be specified as predicate directives. Exemplary directives from  
18 directive types 1212-1220 are illustrated in Tables 1-5. Processing of the script is  
19 then complete.

20 Thus, the present process for applying types and constraints within an  
21 interpretive environment, allows system administrators to easily specify a type,  
22 specify validation requirements, and the like without having to write the  
23 underlying logic for performing this processing. The following is an example of  
24 the constraint processing performed on a command string specified as follows:

25 [Integer][ValidationRange(3,5)]\$a=4.

There are two constraints specified via attribution tokens denoted by “[ ]”. The first attribution token indicates that the variable is a type integer and a second attribution token indicates that the value of the variable \$a must be between 3 and 5 inclusive. The example command string ensures that if the variable \$a is assigned in a subsequent command string or line, the variable \$a will be checked against the two constraints. Thus, the following command strings would each result in an error:

\$a = 231

\$a = “apple”

\$a = \$(get/location).

The constraints are applied at various stages within the administrative tool framework. For example, applicability directives, documentation directives, and parsing guideline directives are processed at a very early stage within the parser. Data generation directives and validation directives are processed in the engine once the parser has finished parsing all the input parameters.

The following tables illustrate representative directives for the various categories, along with an explanation of the processing performed by the administrative tool environment in response to the directive.

Name	Description
PrerequisiteMachineRoleAttribute	Informs shell whether element is to be used only in certain machine roles (e.g., File Server, Mail Server).



PrerequisiteUserRoleAttribute	Informs shell whether element is to be used only in certain user roles (e.g., Domain Administrator, Backup Operator).
PrerequisiteScriptAttribute	Informs the shell this script will be run before excuting the actual command or parameter. Can be used for parameter validation
PrerequisiteUITypeAttribute	This is used to check the User interface available before excuting

Table 1. Applicability Directives

Name	Description
ParsingParameterPositionAttribute	Maps unqualified parameters based on position.
ParsingVariableLengthParameterListAttribute	Maps parameters not having a Parsing ParameterPosition attribute.
ParsingDisallowInteractionAttribute	Specifies action when number of

		parameters is less than required number.
	ParsingRequireInteractionAttribute	Specifies that parameters are obtained through interaction.
	ParsingHiddenElementAttribute	Makes parameter invisible to end user.
	ParsingMandatoryParameterAttribute	Specifies that the parameter is required.
	ParsingPasswordParameterAttribute	Requires special handling of parameter.
	ParsingPromptStringAttribute	Specifies a prompt for the parameter.
	ParsingDefaultAnswerAttribute	Specifies default answer for parameter.
	ParsingDefaultAnswerScriptAttribute	Specifies action to get default answer for parameter.
	ParsingDefaultValueAttribute	Specifies default value for parameter.
	ParsingDefaultValueScriptAttribute	Specifies action to get default value for parameter.
	ParsingParameterMappingAttribute	Specifies a way to

	group parameters
ParsingParameterDeclarationAttribute	This defines that the filed is a parameter
ParsingAllowPipelineInputAttribute	Defines the parameter can be populated from the pipeline

Table 2. Parsing Guideline Directives

Name	Description
DocumentNameAttribute	Provides a Name to refer to elements for interaction or help.
DocumentShortDescriptionAttribute	Provides brief description of element.
DocumentLongDescriptionAttribute	Provides detailed description of element.
DocumentExampleAttribute	Provides example of element.
DocumentSeeAlsoAttribute	Provides a list of related elements.
DocumentSynopsisAttribute	Provides documentation information for element.

Table 3. Documentation Directives

1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14  
15  
16  
17  
18  
19  
20  
21  
22  
23  
24  
25

Name	Description
ValidationRangeAttribute	Specifies that parameter must be within certain range.
ValidationSetAttribute	Specifies that parameter must be within certain collection.
ValidationPatternAttribute	Specifies that parameter must fit a certain pattern.
ValidationLengthAttribute	Specifies the strings must be within size range.
ValidationTypeAttribute	Specifies that parameter must be of certain type.
ValidationCountAttributue	Specifies that input items must be of a certain number.
ValidationFileAttribute	Specifies certain properties for a file.
ValidationFileAttributesAttribute	Specifies certain properties for a file.
ValidationFileSizeAttribute	Specifies that files must be within specified range.
ValidationNetworkAttribute	Specifies that given Network Entity supports certain properties.
ValidationScriptAttribute	Specifies conditions to evaluate

	before using element.
ValidationMethodAttribute	Specifies conditions to evaluate before using element.

Table 4. Data Validation Directives

Name	Description
ProcessingTrimStringAttribute	Specifies size limit for strings.
ProcessingTrimCollectionAttribute	Specifies size limit for collection.
EncodingTypeCoercionAttribute	Specifies Type that objects are to be encoded.
ExpansionWildcardsAttribute	Provides a mechanism to allow globbing

Table 5. Processing and Encoding Directives

When the exemplary administrative tool framework is operating within the .NET™ Framework, each category has a base class that is derived from a basic category class (e.g., CmdAttribute). The basic category class derives from a System.Attribute class. Each category has a pre-defined function (e.g., attrib.func() ) that is called by the parser during category processing. The script author may create a custom category that is derived from a custom category class

1 (e.g., CmdCustomAttribute). The script author may also extend an existing  
2 category class by deriving a directive class from the base category class for that  
3 category and override the pre-defined function with their implementation. The  
4 script author may also override directives and add new directives to the pre-  
5 defined set of directives.

6 The order of processing of these directives may be stored in an external  
7 data store accessible by the parser. The administrative tool framework looks for  
8 registered categories and calls a function (e.g., ProcessCustomDirective) for each  
9 of the directives in that category. Thus, the order of category processing may be  
10 dynamic by storing the category execution information in a persistent store. At  
11 different processing stages, the parser checks in the persistent store to determine if  
12 any metadata category needs to be executed at that time. This allows categories to  
13 be easily deprecated by removing the category entry from the persistent store.

#### 14 **Exemplary Processing of Command Strings**

15 One exemplary process for processing command strings is now described.  
16 FIGURE 13 is a functional flow diagram graphically illustrating the processing of  
17 a command string 1350 through a parser 220 and a core engine 224 within the  
18 administrative tool framework shown in FIGURE 2. The exemplary command  
19 string 1350 pipelines several commands (i.e., process command 1360, where  
20 command 1362, sort command 1364, and table command 1366). The command  
21 line 1350 may pass input parameters to any of the commands (e.g., "handlecount >  
22 400" is passed to the where command 1362). One will note that the process  
23 command 1360 does not have any associated input parameters.  
24  
25

1 In the past, each command was responsible for parsing the input parameters  
2 associated with the command, determining whether the input parameters were  
3 valid, and issuing error messages if the input parameters were not valid. Because  
4 the commands were typically written by various programmers, the syntax for the  
5 input parameters on the command line was not very consistent. In addition, if an  
6 error occurred, the error message, even for the same error, was not very consistent  
7 between the commands.

8 For example, in a UNIX environment, an "ls" command and a "ps"  
9 command have many inconsistencies between them. While both accept an option  
10 "-w", the "-w" option is used by the "ls" command to denote the width of the page,  
11 while the "-w" option is used by the "ps" command to denote print wide output (in  
12 essence, ignoring page width). The help pages associated with the "ls" and the  
13 "ps" command have several inconsistencies too, such as having options bolded in  
14 one and not the other, sorting options alphabetically in one and not the other,  
15 requiring some options to have dashes and some not.

16 The present administrative tool framework provides a more consistent  
17 approach and minimizes the amount of duplicative code that each developer must  
18 write. The administrative tool framework 200 provides a syntax (e.g., grammar), a  
19 corresponding semantics (e.g., a dictionary), and a reference model to enable  
20 developers to easily take advantage of common functionality provided by the  
21 administrative tool framework 200.

22 Before describing the present invention any further, definitions for  
23 additional terms appearing through-out this specification are provided. Input  
24 parameter refers to input-fields for a cmdlet. Argument refers to an input  
25

1 parameter passed to a cmdlet that is the equivalent of a single string in the argv  
2 array or passed as a single element in a cmdlet object. As will be described below,  
3 a cmdlet provides a mechanism for specifying a grammar. The mechanism may  
4 be provided directly or indirectly. An argument is one of an option, an option-  
5 argument, or an operand following the command-name. Examples of arguments  
6 are given based on the following command line:

7  
8 findstr /i /d:\winnt;\winnt\system32 aa\*b \*.ini.  
9

10  
11 In the above command line, "findstr" is argument 0, "/i" is argument 1,  
12 "/d:\winnt;\winnt\system32" is argument 2, "aa\*b" is argument 3, and "\*.ini" is  
13 argument 4. An "option" is an argument to a cmdlet that is generally used to  
14 specify changes to the program's default behavior. Continuing with the example  
15 command line above, "/i" and "/d" are options. An "option-argument" is an input  
16 parameter that follows certain options. In some cases, an option-argument is  
17 included within the same argument string as the option. In other cases, the option-  
18 argument is included as the next argument. Referring again to the above  
19 command line, "winnt;\winnt\system32" is an option-argument. An "operand" is  
20 an argument to a cmdlet that is generally used as an object supplying information  
21 to a program necessary to complete program processing. Operands generally  
22 follow the options in a command line. Referring to the example command line  
23 above again, "aa\*b" and "\*.ini" are operands. A "parsable stream" includes the  
24 arguments.  
25



1 Referring to FIGURE 13, parser **220** parses a parsable stream (e.g.,  
2 command string **1350**) into constituent parts **1320-1326** (e.g., where portion **1322**).  
3 Each portion **1320-1326** is associated with one of the cmdlets **1330-1336**. Parser  
4 **220** and engine **224** perform various processing, such as parsing, parameter  
5 validation, data generation, parameter processing, parameter encoding, and  
6 parameter documentation. Because parser **220** and engine **224** perform common  
7 functionality on the input parameters on the command line, the administrative tool  
8 framework **200** is able to issue consistent error messages to users.

9 As one will recognize, the executable cmdlets **1330-1336** written in  
10 accordance with the present administrative tool framework require less code than  
11 commands in prior administrative environments. Each executable cmdlet **1330-**  
12 **1336** is identified using its respective constituent part **1320-1326**. In addition,  
13 each executable cmdlet **1330-1336** outputs objects (represented by arrows **1340**,  
14 **1342**, **1344**, and **1346**) which are input as input objects (represented by arrows  
15 **1341**, **1343**, and **1345**) to the next pipelined cmdlet. These objects may be input  
16 by passing a reference (e.g., handle) to the object. The executable cmdlets **1330-**  
17 **1336** may then perform additional processing on the objects that were passed in.

18 FIGURE 14 is a logical flow diagram illustrating in more detail the  
19 processing of command strings suitable for use within the process for handling  
20 input shown in FIGURE 9. The command string processing begins at block **1401**,  
21 where either the parser or the script engine identified a command string within the  
22 input. In general the core engine performs set-up and sequencing of the data flow  
23 of the cmdlets. The set-up and sequencing for one cmdlet is described below, but  
24 is applicable to each cmdlet in a pipeline. Processing continues at block **1404**.  
25

1       At block **1404**, a cmdlet is identified. The identification of the cmdlet may  
2 be thru registration. The core engine determines whether the cmdlet is local or  
3 remote. The cmdlet may execute in the following locations: 1) within the  
4 application domain of the administrative tool framework; 2) within another  
5 application domain of the same process as the administrative tool framework; 3)  
6 within another process on the same computing device; or 4) within a remote  
7 computing device. The communication between cmdlets operating within the  
8 same process is through objects. The communication between cmdlets operating  
9 within different processes is through a serialized structured data format. One  
10 exemplary serialized structured data format is based on the extensible markup  
11 language (XML). Processing continues at block **1406**.

12       At block **1406**, an instance of the cmdlet object is created. An exemplary  
13 process for creating an instance of the cmdlet is described below in conjunction  
14 with FIGURE 15. Once the cmdlet object is created, processing continues at  
15 block **1408**.

16       At block **1408**, the properties associated with the cmdlet object are  
17 populated. As described above, the developer declares properties within a cmdlet  
18 class or within an external source. Briefly, the administrative tool framework will  
19 decipher the incoming object(s) to the cmdlet instantiated from the cmdlet class  
20 based on the name and type that is declared for the property. If the types are  
21 different, the type may be coerced via the extended data type manager. As  
22 mentioned earlier, in pipelined command strings, the output of each cmdlet may be  
23 a list of handles to objects. The next cmdlet may inputs this list of object handles,  
24 performs processing, and passes another list of object handles to the next cmdlet.  
25 In addition, as illustrated in FIGURE 7, input parameters may be specified as

1 coming from the command line. One exemplary method for populating properties  
2 associated with a cmdlet is described below in conjunction with FIGURE 16.  
3 Once the cmdlet is populated, processing continues at block 1410.

4 At block 1410, the cmdlet is executed. In overview, the processing  
5 provided by the cmdlet is performed at least once, which includes processing for  
6 each input object to the cmdlet. Thus, if the cmdlet is the first cmdlet within a  
7 pipelined command string, the processing is executed once. For subsequent  
8 cmdlets, the processing is executed for each object that is passed to the cmdlet.  
9 One exemplary method for executing cmdlets is described below in conjunction  
10 with FIGURE 5. When the input parameters are only coming from the command  
11 line, execution of the cmdlet uses the default methods provided by the base cmdlet  
12 case. Once the cmdlet is finished executing, processing proceeds to block 1412.

13 At block 1412, the cmdlet is cleaned-up. This includes calling the  
14 destructor for the associated cmdlet object which is responsible for de-allocating  
15 memory and the like. The processing of the command string is then complete.

#### 16 Exemplary Process for Creating a Cmdlet Object

17 FIGURE 15 is a logical flow diagram illustrating an exemplary process for  
18 creating a cmdlet object suitable for use within the processing of command strings  
19 shown in FIGURE 14. At this point, the cmdlet data structure has been developed  
20 and attributes and expected input parameters have been specified. The cmdlet has  
21 been compiled and has been registered. During registration, the class name (i.e.,  
22 cmdlet name) is written in the registration store and the metadata associated with  
23 the cmdlet has been stored. The process 1500 begins at block 1501, where the  
24 parser has received input (e.g., keystrokes) indicating a cmdlet. The parser may  
25 recognize the input as a cmdlet by looking up the input from within the registry

1 and associating the input with one of the registered cmdlets. Processing proceeds  
2 to block **1504**.

3       At block **1504**, metadata associated with the cmdlet object class is read.  
4 The metadata includes any of the directives associated with the cmdlet. The  
5 directives may apply to the cmdlet itself or to one or more of the parameters.  
6 During cmdlet registration, the registration code registers the metadata into a  
7 persistent store. The metadata may be stored in an XML file in a serialized  
8 format, an external database, and the like. Similar to the processing of directives  
9 during script processing, each category of directives is processed at a different  
10 stage. Each metadata directive handles its own error handling. Processing  
11 continues at block **1506**.

12       At block **1506**, a cmdlet object is instantiated based on the identified cmdlet  
13 class. Processing continues at block **1508**.

14       At block **1508**, information is obtained about the cmdlet. This may occur  
15 through reflection or other means. The information is about the expected input  
16 parameters. As mentioned above, the parameters that are declared public (e.g.,  
17 public string Name **730**) correspond to expected input parameters that can be  
18 specified in a command string on a command line or provided in an input stream.  
19 The administrative tool framework through the extended type manager, described  
20 in FIGURE 18, provides a common interface for returning the information (on a  
21 need basis) to the caller. Processing continues at block **1510**.

22       At block **1510**, applicability directives (e.g., Table 1) are applied. The  
23 applicability directives insure that the class is used in certain machine roles and/or  
24 user roles. For example, certain cmdlets may only be used by Domain  
25

1 Administrators. If the constraint specified in one of the applicability directives is  
2 not met, an error occurs. Processing continues at block **1512**.

3 At block **1512**, metadata is used to provide intellisense. At this point in  
4 processing, the entire command string has not yet been entered. The  
5 administrative tool framework, however, knows the available cmdlets. Once a  
6 cmdlet has been determined, the administrative tool framework knows the input  
7 parameters that are allowed by reflecting on the cmdlet object. Thus, the  
8 administrative tool framework may auto-complete the cmdlet once a  
9 disambiguating portion of the cmdlet name is provided, and then auto-complete  
10 the input parameter once a disambiguating portion of the input parameter has been  
11 typed on the command line. Auto-completion may occur as soon as the portion of  
12 the input parameter can identify one of the input parameters unambiguously. In  
13 addition, auto-completion may occur on cmdlet names and operands too.  
14 Processing continues at block **1514**.

15 At block **1514**, the process waits until the input parameters for the cmdlet  
16 have been entered. This may occur once the user has indicated the end of the  
17 command string, such as by hitting a return key. In a script, a new line indicates  
18 the end of the command string. This wait may include obtaining additional  
19 information from the user regarding the parameters and applying other directives.  
20 When the cmdlet is one of the pipelined parameters, processing may begin  
21 immediately. Once, the necessary command string and input parameters have  
22 been provided, processing is complete.

23 Exemplary Process for Populating the Cmdlet  
24  
25

1       An exemplary process for populating a cmdlet is illustrated in FIGURE 16  
2 and is now described, in conjunction with FIGURE 5. In one exemplary  
3 administrative tool framework, the core engine performs the processing to  
4 populate the parameters for the cmdlet. Processing begins at block 1601 after an  
5 instance of a cmdlet has been created. Processing continues to block 1602.

6       At block 1602, a parameter (e.g., ProcessName) declared within the cmdlet  
7 is retrieved. Based on the declaration with the cmdlet, the core engine recognizes  
8 that the incoming input objects will provide a property named "ProcessName". If  
9 the type of the incoming property is different than the type specified in the  
10 parameter declaration, the type will be coerced via the extended type manager.  
11 The process of coercing data types is explained below in the subsection entitled  
12 "Exemplary Extended Type Manager Processing." Processing continues to block  
13 1603.

14       At block 1603, an attribute associated with the parameter is obtained. The  
15 attribute identifies whether the input source for the parameter is the command line  
16 or whether it is from the pipeline. Processing continues to decision block 1604.

17       At decision block 1604, a determination is made whether the attribute  
18 specifies the input source as the command line. If the input source is the  
19 command line, processing continues at block 1609. Otherwise, processing  
20 continues at decision block 1605.

21       At decision block 1605, a determination is made whether the property name  
22 specified in the declaration should be used or whether a mapping for the property  
23 name should be used. This determination is based on whether the command input  
24 specified a mapping for the parameter. The following line illustrates an exemplary  
25

1 mapping of the parameter “ProcessName” to the “foo” member of the incoming  
2 object:

3       \$ get/process | where han\* -gt 500 | stop/process -ProcessName<-foo.

4 Processing continues at block **1606**.

5       At block **1606**, the mapping is applied. The mapping replaces the name of  
6 the expected parameter from “ProcessName” to “foo”, which is then used by the  
7 core engine to parse the incoming objects and to identify the correct expected  
8 parameter. Processing continues at block **1608**.

9       At block **1608**, the extended type manager is queried to locate a value for  
10 the parameter within the incoming object. As explain in conjunction with the  
11 extended type manager, the extended type manager takes the parameter name and  
12 uses reflection to identify a parameter within the incoming object with parameter  
13 name. The extended type manager may also perform other processing for the  
14 parameter, if necessary. For example, the extended type manager may coerce the  
15 type of data to the expected type of data through a conversion mechanism  
16 described above. Processing continues to decision block **1610**.

17       Referring back to block 1609, if the attribute specifies that the input source  
18 is the command line, data from the command line is obtained. Obtaining the data  
19 from the command line may be performed via the extended type manager.  
20 Processing then continues to decision block 1610.

21       At decision block **1610**, a determination is made whether there is another  
22 expected parameter. If there is another expected parameter, processing loops back  
23 to block **1602** and proceeds as described above. Otherwise, processing is  
24 complete and returns.

1        Thus, as shown, cmdlets act as a template for shredding incoming data to  
2 obtain the expected parameters. In addition, the expected parameters are obtained  
3 without knowing the type of incoming object providing the value for the expected  
4 parameter. This is quite different than traditional administrative environments.  
5 Traditional administrative environments are tightly bound and require that the type  
6 of object be known at compile time. In addition, in traditional environments, the  
7 expected parameter would have been passed into the function by value or by  
8 reference. Thus, the present parsing (e.g., “shredding”) mechanism allows  
9 programmers to specify the type of parameter without requiring them to  
10 specifically know how the values for these parameters are obtained.

11        For example, given the following declaration for the cmdlet Foo:

```
12  
13        class Foo : Cmdlet  
14  
15        {  
16  
17                string Name;  
18  
19                Bool Recurse;  
20  
21        }  
22
```

23        The command line syntax may be any of the following:

24        \$ Foo -Name: (string) -Recurse: True

25        \$ Foo -Name <string> -Recurse True



1       \$Foo -Name (string).

2  
3       The set of rules may be modified by system administrators in order to yield  
4 a desired syntax. In addition, the parser may support multiple sets of rules, so that  
5 more than one syntax can be used by users. In essence, the grammar associated  
6 with the cmdlet structure (e.g., string Name and Bool Recurse) drives the parser.

7  
8       In general, the parsing directives describe how the parameters entered as  
9 the command string should map to the expected parameters identified in the  
10 cmdlet object. The input parameter types are checked to determine whether  
11 correct. If the input parameter types are not correct, the input parameters may be  
12 coerced to become correct. If the input parameter types are not correct and can not  
13 be coerced, a usage error is printed. The usage error allows the user to become  
14 aware of the correct syntax that is expected. The usage error may obtain  
15 information describing the syntax from the Documentation Directives. Once the  
16 input parameter types have either been mapped or have been verified, the  
17 corresponding members in the cmdlet object instance are populated. As the  
18 members are populated, the extended type manager provides processing of the  
19 input parameter types. Briefly, the processing may include a property path  
20 mechanism, a key mechanism, a compare mechanism, a conversion mechanism, a  
21 globber mechanism, a relationship mechanism, and a property set mechanism.  
22 Each of these mechanisms is described in detail below in the section entitled  
23 “Extended Type Manager Processing”, which also includes illustrative examples.

24       Exemplary Process for Executing the Cmdlet

1       An exemplary process for executing a cmdlet is illustrated in FIGURE 17  
2 and is now described. In one exemplary administrative tool environment, the core  
3 engine executes the cmdlet. As mentioned above, the code **1442** within the second  
4 method **1440** is executed for each input object. Processing begins at block **1701**  
5 where the cmdlet has already been populated. Processing continues at block **1702**.

6       At block **1702**, a statement from the code **542** is retrieved for execution.  
7 Processing continues at decision block **1704**.

8       At decision block **1704**, a determination is made whether a hook is included  
9 within the statement. Turning briefly to FIGURE 5, the hook may include calling  
10 an API provided by the core engine. For example, statement **550** within the code  
11 **542** of cmdlet **500** in FIGURE 5 calls the confirmprocessing API specifying the  
12 necessary parameters, a first string (e.g., "PID="), and a parameter (e.g., PID).  
13 Turning back to FIGURE 17, if the statement includes the hook, processing  
14 continues to block **1712**. Thus, if the instruction calling the confirmprocessing  
15 API is specified, the cmdlet operates in an alternate executing mode that is  
16 provided by the operating environment. Otherwise, processing continues at block  
17 **1706** and execution continues in the "normal" mode.

18       At block **1706**, the statement is processed. Processing then proceeds to  
19 decision block **1708**. At block **1708**, a determination is made whether the code  
20 includes another statement. If there is another statement, processing loops back to  
21 block **1702** to get the next statement and proceeds as described above. Otherwise,  
22 processing continues to decision block **1714**.

23       At decision block **1714**, a determination is made whether there is another  
24 input object to process. If there is another input object, processing continues to  
25 block **1716** where the cmdlet is populated with data from the next object. The

1 population process described in FIGURE 16 is performed with the next object.  
2 Processing then loops back to block 1702 and proceeds as described above. Once  
3 all the objects have been processed, the process for executing the cmdlet is  
4 complete and returns.

5 Returning back to decision block 1704, if the statement includes the hook,  
6 processing continues to block 1712. At block 1712, the additional features  
7 provided by the administrative tool environment are processed. Processing  
8 continues at decision block 1708 and continues as described above.

9 The additional processing performed within block 1712 is now described in  
10 conjunction with the exemplary data structure 600 illustrated in FIGURE 6. As  
11 explained above, within the command base class 600 there may be parameters  
12 declared that correspond to additional expected input parameters (e.g., a switch).

13 The switch includes a predetermined string, and when recognized, directs  
14 the core engine to provide additional functionality to the cmdlet. If the parameter  
15 verbose 610 is specified in the command input, verbose statements 614 are  
16 executed. The following is an example of a command line that includes the  
17 verbose switch:

18  
19 \$ get/process | where "han\* -gt 500" | stop/process -verbose.  
20

21 In general, when "-verbose" is specified within the command input, the  
22 core engine executes the command for each input object and forwards the actual  
23 command that was executed for each input object to the host program for display.  
24 The following is an example of output generated when the above command line is  
25 executed in the exemplary administrative tool environment:

1  
2       \$ stop/process PID=15

3       \$ stop/process PID=33.  
4

5       If the parameter whatif **620** is specified in the command input, whatif  
6 statements **624** are executed. The following is an example of a command line that  
7 includes the whatif switch:  
8

9       \$ get/process | where "han\* -gt 500" | stop/process -whatif.  
10

11       In general, when "-whatif" is specified, the core engine does not actually  
12 execute the code **542**, but rather sends the commands that would have been  
13 executed to the host program for display. The following is an example of output  
14 generated when the above command line is executed in the administrative tool  
15 environment of the present invention:  
16

17       #\$ stop/process PID=15

18       #\$ stop/process PID=33.  
19

20       If the parameter confirm **630** is specified in the command input, confirm  
21 statements **634** are executed. The following is an example of a command line that  
22 includes the confirm switch:  
23

24       \$ get/process | where "han\* -gt 500" | stop/process -confirm.  
25

1 In general, when “-confirm” is specified, the core engine requests  
2 additional user input on whether to proceed with the command or not. The  
3 following is an example of output generated when the above command line is  
4 executed in the administrative tool environment of the present invention.

5  
6 \$ stop/process PID 15

7 Y/N Y

8 \$ stop/process PID 33

9 Y/N N.  
10

11 As described above, the exemplary data structure **600** may also include a  
12 security method **640** that determines whether the task being requested for  
13 execution should be allowed. In traditional administrative environments, each  
14 command is responsible for checking whether the person executing the command  
15 has sufficient privileges to perform the command. In order to perform this check,  
16 extensive code is needed to access information from several sources. Because of  
17 these complexities, many commands did not perform a security check. The  
18 inventors of the present administrative tool environment recognized that when the  
19 task is specified in the command input, the necessary information for performing  
20 the security check is available within the administrative tool environment.  
21 Therefore, the administrative tool framework performs the security check without  
22 requiring complex code from the tool developers. The security check may be  
23 performed for any cmdlet that defines the hook within its cmdlet. Alternatively,  
24 the hook may be an optional input parameter that can be specified in the command  
25 input, similar to the verbose parameter described above.

1       The security check is implemented to support roles based authentication,  
2       which is generally defined as a system of controlling which users have access to  
3       resources based on the role of the user. Thus, each role is assigned certain access  
4       rights to different resources. A user is then assigned to one or more roles. In  
5       general, roles based authentication focus on three items: principle, resource, and  
6       action. The *principle* identifies who requested the *action* to be performed on the  
7       *resource*.

8       The inventors of the present invention recognized that the cmdlet being  
9       requested corresponded to the action that was to be performed. In addition, the  
10      inventors appreciated that the owner of the process in which the administrative  
11      tool framework was executing corresponded to the principle. Further, the  
12      inventors appreciated that the resource is specified within the cmdlet. Therefore,  
13      because the administrative tool framework has access to these items, the inventors  
14      recognized that the security check could be performed from within the  
15      administrative tool framework without requiring tool developers to implement the  
16      security check.

17      The operation of the security check may be performed any time additional  
18      functionality is requested within the cmdlet by using the hook, such as the  
19      confirmprocessing API. Alternatively, security check may be performed by  
20      checking whether a security switch was entered on the command line, similar to  
21      verbose, whatif, and confirm. For either implementation, the checkSecurity  
22      method calls an API provided by a security process (not shown) that provides a set  
23      of APIs for determining who is allowed. The security process takes the  
24      information provided by the administrative tool framework and provides a result  
25

1 indicating whether the task may be completed. The administrative tool framework  
2 may then provide an error or just stop the execution of the task.

3 Thus, by providing the hook within the cmdlet, the developers may use  
4 additional processing provided by the administrative tool framework.

### 5 Exemplary Extended Type Manager Processing

6 As briefly mentioned above in conjunction with FIGURE 18, the extended  
7 type manager may perform additional processing on objects that are supplied. The  
8 additional processing may be performed at the request of the parser 220, the script  
9 engine 222, or the pipeline processor 402. The additional processing includes a  
10 property path mechanism, a key mechanism, a compare mechanism, a conversion  
11 mechanism, a globber mechanism, a relationship mechanism, and a property set  
12 mechanism. Those skilled in the art will appreciate that the extended type  
13 manager may also be extended with other processing without departing from the  
14 scope of the claimed invention. Each of the additional processing mechanisms is  
15 now described.

16 First, the property path mechanism allows a string to navigate properties of  
17 objects. In current reflection systems, queries may query properties of an object.  
18 However, in the present extended type manager, a string may be specified that will  
19 provide a navigation path to successive properties of objects. The following is an  
20 illustrative syntax for the property path: P1.P2.P3.P4.

21 Each component (e.g., P1, P2, P3, and P4) comprises a string that may  
22 represent a property, a method with parameters, a method without parameters, a  
23 field, an XPATH, or the like. An XPATH specifies a query string to search for an  
24 element (e.g., "/FOO@=13"). Within the string, a special character may be  
25

1 included to specifically indicate the type of component. If the string does not  
2 contain the special character, the extended type manager may perform a lookup to  
3 determine the type of component. For example, if component P1 is an object, the  
4 extended type manager may query whether P2 is a property of the object, a  
5 method on the object, a field of the object, or a property set. Once the extended  
6 type manager identifies the type for P2, processing according to that type is  
7 performed. If the component is not one of the above types, the extended type  
8 manager may further query the extended sources to determine whether there is a  
9 conversion function to convert the type of P1 into the type of P2. These and other  
10 lookups will now be described using illustrative command strings and showing the  
11 respective output.

12 The following is an illustrative string that includes a property path:

13 \$ get/process | /where hand\* -gt 500 | format/table name.toupper, ws.kb,  
14 exe\*.ver\*.description.tolower.trunc(30).

15  
16 In the above illustrative string, there are three property paths: (1)  
17 “name.toupper”; (2) “ws.kb”; and (3) “exe\*.ver\*.description.tolower.trunc(30).  
18 Before describing these property paths, one should note that “name”, “ws”, and  
19 “exe” specify the properties for the table. In addition, one should note that each of  
20 these properties is a direct property of the incoming object, originally generated by  
21 “get/process” and then pipelined through the various cmdlets. Processing  
22 involved for each of the three property paths will now be described.

23 In the first property path (i.e., “name.toupper”), name is a direct property of  
24 the incoming object and is also an object itself. The extended type manager  
25 queries the system using the priority lookup described above to determine the type



1 for toupper. The extended type manager discovers that toupper is not a property.  
2 However, toupper may be a method inherited by a string type to convert lower  
3 case letters to upper case letters within the string. Alternatively, the extended type  
4 manager may have queried the extended metadata to determine whether there is  
5 any third party code that can convert a name object to upper case. Upon finding  
6 the component type, processing is performed in accordance with that component  
7 type.

8 In the second property path (i.e., "ws.kb"), "ws" is a direct property of the  
9 incoming object and is also an object itself. The extended type manager  
10 determines that "ws" is an integer. Then, the extended type manager queries  
11 whether kb is a property of an integer, whether kb is a method of an integer, and  
12 finally queries whether any code knows how to take an integer and convert the  
13 integer to a kb type. Third party code is registered to perform this conversion and  
14 the conversion is performed.

15 In the third property path (i.e., "exe\*.ver\*.description.tolower.trunc(30)"),  
16 there are several components. The first component ("exe\*") is a direct property of  
17 the incoming object and is also an object. Again, the extended type manager  
18 proceeds down the lookup query in order to process the second component  
19 ("ver\*"). The "exe\*" object does not have a "ver\*" property or method, so the  
20 extend type manager queries the extended metadata to determine whether there is  
21 any code that is registered to convert an executable name into a version. For this  
22 example, such code exists. The code may take the executable name string and use  
23 it to open a file, then accesses the version block object, and return the description  
24 property (the third component ("description") of the version block object. The  
25

extended type manager then performs this same lookup mechanism for the fourth component ("tolower") and the fifth component ("trunc(40)"). Thus, as illustrated, the extended type manager may perform quite elaborate processing on a command string without the administrator needing to write any specific code. Table 1 illustrates output generated for the illustrative string.

Name.toupper ws.kb exe\*.ver\*.description.tolower.trunc(30)

ETCLIENT 29,964 etclient

CSRSS 6,944

SVCHOST 28,944 generic host process for win32

OUTLOOK 18,556 office outlook

MSMSGSGS 13,248 messenger

Table 1.

Another query mechanism 1824 includes a key. The key identifies one or more properties that make an instance of the data type unique. For example, in a database, one column may be identified as the key which can uniquely identify each row (e.g., social security number). The key is stored within the type metadata 1840 associated with the data type. This key may then be used by the extended type manager when processing objects of that data type. The data type may be an extended data type or an existing data type.

Another query mechanism 1824 includes a compare mechanism. The compare mechanism compares two objects. If the two objects directly support the compare function, the directly supported compare function is executed. However, if neither object supports a compare function, the extended type manager may look in the type metadata for code that has been registered to support the compare

1 between the two objects. An illustrative series of command line strings invoking  
2 the compare mechanism is shown below, along with corresponding output in  
3 Table 2.

4  
5 \$ \$a = \$( get/date )

6 \$ start/sleep 5

7 \$ \$b = \$( get/date

8 compare/time \$a \$b

9  
10 Ticks : 51196579

11 Days : 0

12 Hours : 0

13 Milliseconds : 119

14 Minutes : 0

15 Seconds : 5

16 TotalDays : 5.92552997685185E-05

17 TotalHours : 0.00142212719444444

18 TotalMilliseconds : 5119.6579

19 TotalMinutes : 0.0853276316666667

20 TotalSeconds : 5.1196579

21  
22 Table 2.

23  
24 Compare/time cmdlet is written to compare two datetime objects. In this  
25 case, the DateTime object supports the IComparable interface.

1 Another query mechanism 1824 includes a conversion mechanism. The  
2 extended type manager allows code to be registered stating its ability to perform a  
3 specific conversion. Then, when an object of type A is input and a cmdlet  
4 specifies an object of type B, the extended type manager may perform the  
5 conversion using one of the registered conversions. The extended type manager  
6 may perform a series of conversions in order to coerce type A into type B. The  
7 property path described above (“ws.kb”) illustrates a conversion mechanism.

8 Another query mechanism 1824 includes a globber mechanism. A globber  
9 refers to a wild card character within a string. The globber mechanism inputs the  
10 string with the wild card character and produces a set of objects. The extended  
11 type manager allows code to be registered that specifies wildcard processing. The  
12 property path described above (“exe\*.ver\*.description.tolower.trunc(30)”) illustrates the globber mechanism. A registered process may provide globbing for  
13 file names, file objects, incoming properties, and the like.

15 Another query mechanism 1824 includes a property set mechanism. The  
16 property set mechanism allows a name to be defined for a set of properties. An  
17 administrator may then specify the name within the command string to obtain the  
18 set of properties. The property set may be defined in various ways. In one way, a  
19 predefined parameter, such as “?”, may be entered as an input parameter for a  
20 cmdlet. The operating environment upon recognizing the predefined parameter  
21 lists all the properties of the incoming object. The list may be a GUI that allows  
22 an administrator to easily check (e.g., “click on”) the properties desired and name  
23 the property set. The property set information is then stored in the extended  
24  
25

1 metadata. An illustrative string invoking the property set mechanism is shown  
2 below, along with corresponding output in Table 3:

3 \$ get/process | where han\* -gt> 500 | format/table config.

4 In this illustrative string, a property set named “config” has been defined to  
5 include a name property, a process id property (Pid), and a priority property. The  
6 output for the table is shown below.

<u>Name</u>	<u>Pid</u>	<u>Priority</u>
ETClient	3528	Normal
csrss	<b>528</b>	Normal
svchost	<b>848</b>	Normal
OUTLOOK	2,772	Normal
msmsgs	2,584	Normal

7  
8  
9  
10  
11  
12  
13  
14 Table 3.

15 Another query mechanism **1824** includes a relationship mechanism. In  
16 contrast to traditional type systems that support one relationship (i.e., inheritance),  
17 the relationship mechanism supports expressing more than one relationship  
18 between types. Again, these relationships are registered. The relationship may  
19 include finding items that the object consumes or finding the items that consume  
20 the object. The extended type manager may access ontologies that describe  
21 various relationships. Using the extended metadata and the code, a specification  
22 for accessing any ontology service, such as OWL, DAWL, and the like, may be  
23 described. The following is a portion of an illustrative string which utilizes the  
24 relationship mechanism: .OWL:”string”.  
25

1       The “OWL” identifier identifies the ontology service and the “string”  
2 specifies the specific string within the ontology service. Thus, the extended type  
3 manager may access types supplied by ontology services.

#### 4 Exemplary Process for Displaying Command Line Data

5       The present mechanism provides a data driven command line output. The  
6 formatting and outputting of the data is provided by one or more cmdlets in the  
7 pipeline of cmdlets. Typically, these cmdlets are included within the non-  
8 management cmdlets described in conjunction with FIGURE 2 above. The  
9 cmdlets may include a format cmdlet, a markup cmdlet, a convert cmdlet, a  
10 transform cmdlet, and an out cmdlet.

11       FIGURE 19 graphically depicts exemplary sequences 1901-1907 of these  
12 cmdlets within a pipeline. The first sequence **1901** illustrates the out cmdlet **1910**  
13 as the last cmdlet in the pipeline. In the same manner as described above for other  
14 cmdlets, the out cmdlet **1910** accepts a stream of pipeline objects generated and  
15 processed by other cmdlets within the pipeline. However, in contrast to most  
16 cmdlets, the out cmdlet **1910** does not emit pipeline objects for other cmdlets.  
17 Instead, the out cmdlet **1910** is responsible for rendering/displaying the results  
18 generated by the pipeline. Each out cmdlet **1910** is associated with an output  
19 destination, such as a device, a program, and the like. For example, for a console  
20 device, the out cmdlet **1910** may be specified as out/console; for an internet  
21 browser, the out cmdlet **1910** may be specified as out/browser; and for a window,  
22 the out cmdlet **1910** may be specified as out/window. Each specific out cmdlet is  
23 familiar with the capabilities of its associated destination. Locale information  
24 (e.g., date & currency formats) are processed by the out cmdlet 1910, unless a  
25

1 convert cmdlet preceded the out cmdlet in the pipeline. In this situation, the  
2 convert cmdlet processed the local information.

3 Each host is responsible for supporting certain out cmdlets, such as  
4 out/console. The host also supports any destination specific host cmdlet (e.g.,  
5 out/chart that directs output to a chart provided by a spreadsheet application). In  
6 addition, the host is responsible for providing default handling of results. The out  
7 cmdlet in this sequence may decide to implement its behavior by calling other  
8 output processing cmdlets (such as format/markup/convert/transform). Thus, the  
9 out cmdlet may implicitly modify sequence **1901** to any of the other sequences or  
10 may add its own additional format/output cmdlets.

11 The second sequence **1902** illustrates a format cmdlet **1920** before the out  
12 cmdlet **1910**. For this sequence, the format cmdlet **1920** accepts a stream of  
13 pipeline objects generated and processed by other cmdlets within the pipeline. In  
14 overview, the format cmdlet **1920** provides a way to select display properties and a  
15 way to specify a page layout, such as shape, column widths, headers, footers, and  
16 the like. The shape may include a table, a wide list, a columnar list, and the like.  
17 In addition, the format cmdlet **1920** may include computations of totals or sums.  
18 Exemplary processing performed by a format cmdlet **1920** is described below in  
19 conjunction with FIGURE 20. Briefly, the format cmdlet emits format objects, in  
20 addition to emitting pipeline objects. The format objects can be recognized  
21 downstream by an out cmdlet (e.g., out cmdlet **1920** in sequence **1902**) via the  
22 extended type manager or other mechanism. The out cmdlet **1920** may choose to  
23 either use the emitted format objects or may choose to ignore them. The out  
24 cmdlet determines the page layout based on the page layout data specified in the  
25 display information. In certain instances, modifications to the page layout may be

1 specified by the out cmdlet. In one exemplary process the out cmdlet may  
2 determine an unspecified column width by finding a maximum length for each  
3 property of a predetermined number of objects (e.g., 50) and setting the column  
4 width to the maximum length. The format objects include formatting information,  
5 header/footer information, and the like.

6 The third sequence **1903** illustrates a format cmdlet **1920** before the out  
7 cmdlet 1910. However, in the third sequence 1903, a markup cmdlet **1930** is  
8 pipelined between the format cmdlet **1920** and the out cmdlet 1910. The markup  
9 cmdlet **1930** provides a mechanism for adding property annotation (e.g., font,  
10 color) to selected parameters. Thus, the markup cmdlet **1930** appears before the  
11 output cmdlet 1910. The property annotations may be implemented using a  
12 “shadow property bag”, or by adding property annotations in a custom namespace  
13 in a property bag. The markup cmdlet **1930** may appear before the format cmdlet  
14 **1920** as long as the markup annotations may be maintained during processing of  
15 the format cmdlet 1920.

16 The fourth sequence **1904** again illustrates a format cmdlet **1920** before the  
17 out cmdlet 1910. However, in the fourth sequence 1904, a convert cmdlet **1940** is  
18 pipelined between the format cmdlet **1920** and the out cmdlet 1910. The convert  
19 cmdlet **1940** is also configured to process the format objects emitted by the format  
20 cmdlet 1920. The convert cmdlet **1940** converts the pipelined objects into a  
21 specific encoding based on the format objects. The convert cmdlet **1940** is  
22 associated with the specific encoding. For example, the convert cmdlet **1940** that  
23 converts the pipelined objects into Active Directory Objects (ADO) may be  
24 declared as “convert/ADO” on the command line. Likewise, the convert cmdlet  
25 **1940** that converts the pipelined objects into comma separated values (csv) may be



1 declared as “convert/csv” on the command line. Some of the convert cmdlets  
2 **1940** (e.g., convert/XML and convert/html) may be blocking commands, meaning  
3 that all the pipelined objects are received before executing the conversion.  
4 Typically, the out cmdlet **1920** may determine whether to use the formatting  
5 information provided by the format objects. However, when a convert cmdlet  
6 **1920** appears before the out cmdlet 1920, the actual data conversion has already  
7 occurred before the out cmdlet receives the objects. Therefore, in this situation,  
8 the out cmdlet can not ignore the conversion.

9 The fifth sequence **1905** illustrates a format cmdlet 1920, a markup cmdlet  
10 1930, a convert cmdlet 1940, and an out cmdlet **1910** in that order. Thus, this  
11 illustrates that the markup cmdlet **1930** may occur before the convert cmdlet 1940.

12 The sixth sequence **1906** illustrates a format cmdlet 1920, a specific convert  
13 cmdlet (e.g., convert/xml cmdlet 1940’), a specific transform cmdlet (e.g.,  
14 transform/xslt cmdlet 1950), and an out cmdlet 1910. The convert/xml cmdlet  
15 1940’ converts the pipelined objects into an extended markup language (XML)  
16 document. The transform/xslt cmdlet **1950** transforms the XML document into  
17 another XML document using an Extensible Style Language (XSL) style sheet. The  
18 transform process is commonly referred to as extensible style language  
19 transformation (XSLT), in which an XSL processor reads the XML document and  
20 follows the instructions within the XSL style sheet to create the new XML  
21 document.

22 The seventh sequence **1907** illustrates a format cmdlet 1920, a markup  
23 cmdlet 1930, a specific convert cmdlet (e.g., convert/xml cmdlet 1940’), a specific  
24 transform cmdlet (e.g., transform/xslt cmdlet 1950), and an out cmdlet 1910.  
25

1 Thus, the seventh sequence 1907 illustrates having the markup cmdlet 1930  
2 upstream from the convert cmdlet and transform cmdlet.

3 FIGURE 20 illustrates exemplary processing 2000 performed by a format  
4 cmdlet. The formatting process begins at block 2001, after the format cmdlet has  
5 been parsed and invoked by the parser and pipeline processor in a manner  
6 described above. Processing continues at block 2002.

7 At block 2002, a pipeline object is received as input to the format cmdlet.  
8 Processing continues at block 2004.

9 At block 2004, a query is initiated to identify a type for the pipelined  
10 object. This query is performed by the extended type manager as described above  
11 in conjunction with FIGURE 18. Once the extended type manager has identified  
12 the type for the object, processing continues at block 2006.

13 At block 2006, the identified type is looked up in display information. An  
14 exemplary format for the display information is illustrated in FIGURE 21 and will  
15 be described below. Processing continues at decision block 2008.

16 At decision block 2008, a determination is made whether the identified type  
17 is specified within the display information. If there is no entry within the display  
18 information for the identified type, processing is complete. Otherwise, processing  
19 continues at block 2010.

20 At block 2010, formatting information associated with the identified type is  
21 obtained from the display information. Processing continues at block 2012.

22 At block 2012, information is emitted on the pipeline. Once the  
23 information is emitted, the processing is complete.

24 Exemplary information that may be emitted is now described in further  
25 detail. The information may include formatting information, header/footer

1 information, and a group end/begin signal object. The formatting information may  
2 include a shape, a label, numbering/bullets, column widths, character encoding  
3 type, content font properties, page length, group-by-property name, and the like.  
4 Each of these may have additional specifications associated with it. For example,  
5 the shape may specify whether the shape is a table, a list, or the like. Labels may  
6 specify whether to use column headers, list labels, or the like. Character encoding  
7 may specify ASCII, UTF-8, Unicode, and the like. Content font properties may  
8 specify the font that is applied to the property values that are display. A default  
9 font property (e.g., Courier New, 10 point) may be used if content font properties  
10 are not specified.

11 The header/footer information may include a header/footer scope, font  
12 properties, title, subtitle, date, time, page numbering, separator, and the like. For  
13 example, the scope may specify a document, a page, a group, or the like.  
14 Additional properties may be specified for either the header or the footer. For  
15 example, for group and document footers, the additional properties may include  
16 properties or columns to calculate a sum/total, object counts, label strings for totals  
17 and counts, and the like.

18 The group end/begin signal objects are emitted when the format cmdlet  
19 detects that a group-by property has changed. When this occurs, the format cmdlet  
20 treats the stream of pipeline objects as previously sorted and does not re-sort them.  
21 The group end/begin signal objects may be interspersed with the pipeline objects.  
22 Multiple group-by properties may be specified for nested sorting. The format  
23 cmdlet may also emit a format end object that includes final sums and totals.

24 Turning briefly to FIGURE 21, an exemplary display information **2100** is in  
25 a structured format and contains information (e.g., formatting information,

header/footer information, group-by properties or methods) associated with each object that has been defined. For example, the display information 2100 may be XML-based. Each of the afore-mentioned properties may then be specified within the display information. The information within the display information 2100 may be populated by the owner of the object type that is being entered. The operating environment provides certain APIs and cmdlets that allow the owner to update the display information by creating, deleting, and modifying entries.

FIGURE 22 is a table listing an exemplary syntax 2201-2213 for certain format cmdlets (e., format/table, format/list, and format/wide), markup cmdlets (e.g., add/markup), convert cmdlets (e.g., convert/text, convert/sv, convert/csv, convert/ADO, convert/XML, convert/html), transform cmdlets (e.g., transform/XSLT) and out cmdlets (e.g., out/console, out/file). FIGURE 23 illustrates results rendered by the out/console cmdlet using various pipeline sequences of the output processing cmdlets (e.g., format cmdlets, convert cmdlets, and markup cmdlets).

Because the present administrative tool framework allows tasks to be initiated from any host (e.g., the user-interface, the command line, programmatically), system administrators and product developers may share code and experiences which will make both types of users more effective and efficient. In addition, system administrators will only need to learn one set of concepts, commands, and troubleshooting techniques. Likewise, product developers will only need to create one provider cmdlet to leverage all the management tools.

Thus, as described, because the present administrative tool framework simplifies the access to and the control of management objects and provides transparent remoting, the present administrative tool framework decreases the skill

1 level that is required to develop administrative tools. In addition, the framework  
2 allows developers to see the scripts run by GUI interactions which help the  
3 developers learn the framework and to create their own automated scripts. In  
4 addition, because testing has already been performed on the functionality provided  
5 by the framework, product level testing is reduced to only testing the GUI paths to  
6 invoke functions provided by the framework. Another benefit is that the  
7 management GUI can expose its inner workings via cmdlets which provides  
8 developers and testers easy access to the internal state and control of the GUI for  
9 debugging/diagnostics/automated test.

10 Although details of specific implementations and embodiments are  
11 described above, such details are intended to satisfy statutory disclosure  
12 obligations rather than to limit the scope of the following claims. Thus, the  
13 invention as defined by the claims is not limited to the specific features described  
14 above. Rather, the invention is claimed in any of its forms or modifications that  
15 fall within the proper scope of the appended claims, appropriately interpreted in  
16 accordance with the doctrine of equivalents.